CoLink: A Programming Framework for Decentralized Data Science
Milestone Report (Group Name: CoLink Dev)

Xiaoyuan Liu
xiaoyuanliu@berkeley.edu

Tianchen Liu
tianchenliu@berkeley.edu

Bryce Wong
brywong@berkeley.edu

Siyuan Zhuang
source@berkeley.edu

Abstract

Decentralized data collaboration is a common need in many research fields but it faces many deployment challenges, including practicability, compatibility and publicity. In this work, we propose CoLink to simplify the deployment of decentralized data science solutions. As planned in our project proposal, we have finished the first two stages and implemented both the prototype system and the example protocol.

1 Introduction

1.1 Problem Statement

Data collaboration is a common need in many research fields including finance, health care, cybersecurity, and many others. To protect data privacy in such collaborations, many existing protocols and systems support certain types of computations without sharing the original data input across different participants. However, there are many **deployment challenges** for decentralized privacy-preserving solutions in the real world and there lacks a universal framework with a consistent data collaboration workflow. We identify three major types of deployment difficulties.

- **Practicability.** Compared with protocol simulations that focus on correctness and performance evaluation, a real-world privacy-preserving system needs great engineering efforts to be practical. In most solutions, different participants need to be able to discover and identify each other through WAN. They often need to build secure and authenticated connections for communication, data storage and transmission, and performing various computations. Building such a real-world solution requires a system that manages authenticated users and their roles, database, network communication, computation runtime, and many other resources.
- **Compatibility.** The implementation of one protocol often depends on the functionality of many other protocols. For example, federated learning often requires secure aggregation as its building block to merge the gradients in a privacy-preserving way. Secure aggregation might further requires synchronization for randomness. However, different protocols are often implemented with different library dependencies in different programming languages, making it hard for integration. Even protocols implemented in the same programming language might use different data exchange formats. Poor compatibility not only slows down the integration but also makes the upgrade of the protocol difficult, it also discourages developers from replacing one protocol with a similar but better fitting one.
- **Publicity.** Knowing and understanding the terms to describe the protocol requires comprehensive background knowledge in the field of cryptography. Sometimes there are multiple implementations available with various differences that can be only discovered after testing. There lacks a unified way to describe the functionalities and security guarantees of various protocols and an index that supports searching and comparison for them.

1.2 Proposed Approach

We propose to build CoLink, a new programming framework to accelerate the development of security protocols and overcome the deployment obstacles. In most decentralized data collaborations, participants manage their own data as input, communicate with each other, and execute privacy-preserving protocols to get separate output. As shown in Figure 1, we propose to use CoLink to provide a new unified workflow to describe and implement collaborative protocols. In the new workflow, each participant starts a CoLink server in their trusted envi-

ronment and store their private data in it. The CoLink server acts as a data asset manager. To start a protocol, one participant instantiate a task in the system by calling their own CoLink server entrypoint and specifying who they want to collaborate with. Their CoLink server then reaches out representing them to invite other CoLink servers to join the task. Once all participants agree to proceed, the CoLink servers from all participants start the protocol instance for the task.

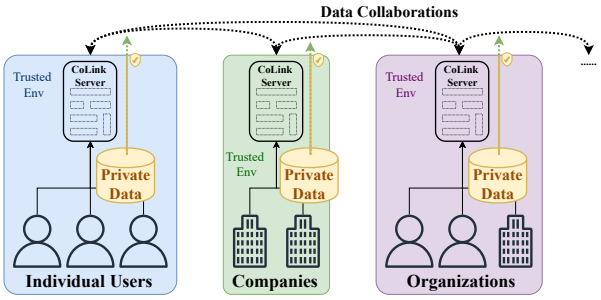

Figure 1: Unified collaboration workflow with CoLink.

To support such a workflow, CoLink provides a unified interface to manage user, storage, communication, and computation. By extending gRPC, we propose to simplify the development of multi-party protocols and orchestrate implementations in different programming languages to work together consistently. With a unified interface that increases potential data contributors, our framework has the potential to enable larger-scale decentralized data collaboration.

Two types of programmers can benefit from CoLink. Data collaboration *application developers* who understand the functionalities and threat models of security protocols can directly integrate off-the-shelf protocols as black boxes into their application system. And *protocol designers* who are often researchers or cryptography experts save efforts on building repeated basic abstractions and writing reusable protocol implementation code.

We design a plug-in architecture for CoLink shown in Figure 2. At a high level, each participant client has a *CoLink server* hosted on their laptop or in a trusted cloud environment and store their data there. CoLink servers manage the storage and data collaboration requests, ask the client to make decisions on whether to approve a collaboration request, and communicate with other CoLink servers to run protocols on behalf of their owner.

Each user's framework contains three components:

- **CoLink Server.** To use CoLink in the programming environment, each client needs to setup, in an environment they trust, a CoLink server that manages the user, storage, and network-related abstractions. For scalability, multiple clients may share the same CoLink server as the server provides user-based isolation. Inside the CoLink server, there is a key-value store for persistent storage, a core scheduler to handle various requests, an inter-core communication module to connect with other CoLink servers, a gRPC service, and a message queue to handle connections from both the client and other CoLink servers with another layer of access control.

- **CoLink SDKs.** To simplify the programming experience for both application developers and protocol designers, we provide SDKs in various programming languages for writing programs that can interact with the CoLink Server. Specifically, for each language, we target to provide two different SDKs: *an application SDK (SDK-A)* for application developers and *a protocol SDK (SDK-P)* for protocol designers. SDK-A allows the client to update storage, manage computation requests, and monitor CoLink server status. And SDK-P allows protocol designers to write CoLink Extensions that extend the functionality of CoLink with new protocols.

- **Protocol Operators (CoLink Extensions).** According to our design principle of being extensible, we detach the implementation of specific protocol from the CoLink server. With SDK-P, a protocol designer can write a program that connects to the CoLink server to register itself as a protocol operator. Whenever someone requests a protocol execution from the CoLink server and gets approved, the CoLink server forwards the request to the corresponding operator.

With the above three components, CoLink provides the necessary programming abstraction to handle computation and can link code in different programming languages from different participants together. By providing SDK-P in different programming languages, CoLink can integrate protocols implemented in different languages as operators. By having SDK-A in different programming languages, the client can use whatever environment they prefer to start certain protocol executions.

1.3 Project Artifacts

In addition to a paper finalizing our design and the presentation, we also target to produce:

1. A prototype CoLink server implemented in Rust. We also target to open-source our solution. We welcome other teams who target protocol design-related research to try out CoLink near the end of the course.
2. An SDK-P in Rust and an SDK-A in both Rust and Python. We will use it to demonstrate the cross-language protocol execution feature.

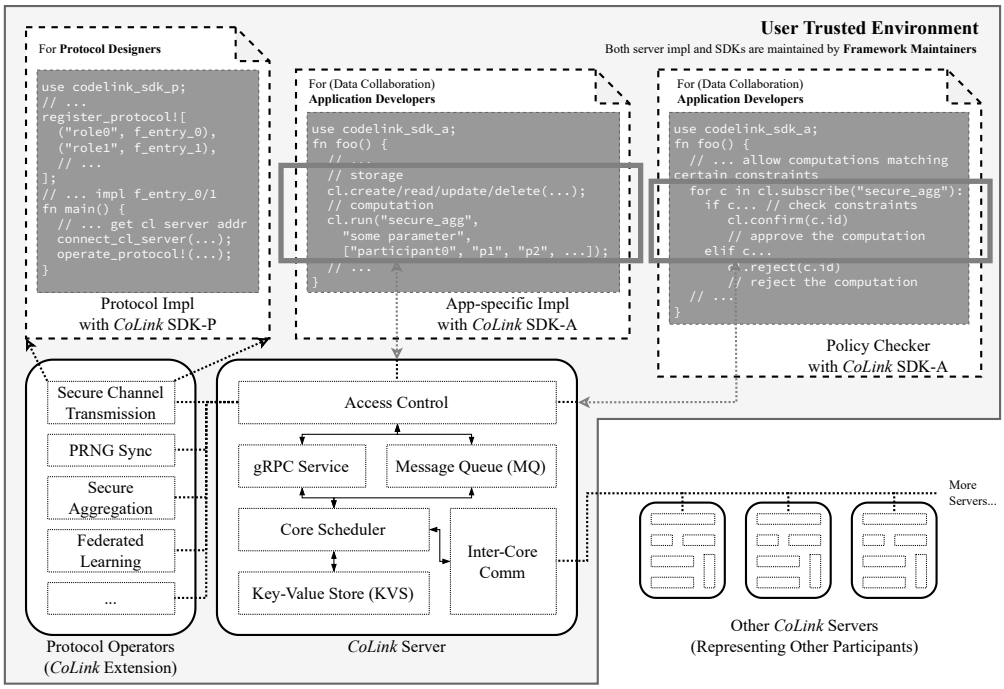

Figure 2: CoLink Architecture.

3. A case study with a basic secure aggregation protocol. We will compare the programming difficulty of writing a basic secure aggregation application with or without CoLink.
4. A project homepage providing tutorial, documentation, and a list of supported protocols for publicity.

2 Progress

As planned in the milestone section in our proposal, we have finalized the core system design, finished an early prototype CoLink server, provided example SDK-A and SDK-P in Rust, and implemented the basic secure aggregation protocol. In addition, we have also started developing a simple frontend to illustrate the core abstractions and have written a short tutorial of implementing secure aggregation from scratch.

2.1 Early Results

To show the simplicity of using CoLink, in Listing 1 ,we demonstrate how to write a simple collaboration protocol under CoLink. As we can see from the example, CoLink provide simple storage-related abstraction (line 29). Protocol designer can integrate their existing work by adding entry functions for each role (line 7 and 21). Serving the function entries with the macro in SDK-P, the code can run as a protocol operator as mentioned in Section 1.2.

Listing 1: CoLink Protocol Example

```
1  #![allow(unused_variables)]
2  use dds_sdk_a::{Dds, Participant};
3  use dds_sdk_p::ProtocolEntry;
4
5  struct Initiator;
6  #[dds_sdk_p::async_trait]
7  impl ProtocolEntry for Initiator {
8      async fn start(
9          &self,
10         dds: Dds,
11         param: Vec<u8>,
12         participants: Vec<Participant>,
13     ) -> Result<(), Box<dyn std::error::Error>> {
14         println!("initiator");
15         Ok(())
16     }
17 }
18
19 struct Receiver;
20 #[dds_sdk_p::async_trait]
21 impl ProtocolEntry for Receiver {
22     async fn start(
23         &self,
24         dds: Dds,
25         param: Vec<u8>,
26         participants: Vec<Participant>,
27     ) -> Result<(), Box<dyn std::error::Error>> {
28         println!("{}", String::from_utf8_lossy(&param));
29         dds.create_entry(&format!("tasks:{}:output",
30               dds.get_task_id()?), &param)
31             .await?;
32         Ok(())
33     }
34 }
35
36 dds_sdk_p::protocol_start!(
37     "greetings",            // protocol_name
38     ("initiator", Initiator), // bind initiator's entry
          function
39     ("receiver", Receiver)  // bind receiver's entry function
);
```

2.2 Expected Challenges

Considering the development of the existing part (hundreds of commits and more than 3000 loc so far) indicates the system can be more complicated than we originally thought. We identify the evaluation of the system to be the major challenge in the next stage. To finish one complete data collaboration, many system components are triggered separately.

Although this is not part of the original proposal, we also plan to integrate existing protocols in the next stage and we believe that can be challenging, especially when the target protocol has its own infrastructure requirement. Some existing protocols have their own network and computation related dependencies which requires careful setup and maintenance.

2.3 Discussion

Based on the team's previous experience in building privacy-preserving systems, we think the current system design will be helpful to both application developers and cryptography protocol designers. Although we do not target to integrate many protocols in the current development stage. For future work, however, we are not certain how hard it would be to integrate existing building blocks (e.g. the ones mentioned in [1, 2]) into our framework. We will continue to evaluate the programming difficulty of integrating various existing efforts during our project development. Our early results in implementing basic secure aggregation under CoLink suggest the current abstractions are easy enough to use but still effective.

To further increase the publicity mentioned in Section 1, in the future we also plan to write CoLink protocol that acts as a data registry. Such a registry can be helpful in the client discovery procedure and further simplify the programming abstraction. We will also explore the possibility of using web3 technology for the registry.

3 Project Milestones & Planned Timeline

We specify the three key stages of the project as follows.

1. **Mar 24.** Finalize system design. Start implementing the CoLink server.
2. **Apr 14.** Finish an early prototype CoLink server, an example SDK, and the basic secure aggregation.
3. **May 5.** Prepare the prototype CoLink. Finish the case study and performance evaluation. After this stage, we will focus on writing the final report.

As also discussed in Section 2.1, we have finished the first two milestones as planned. We expect to finish the third stage also in time.

4 Division of Labor

Xiaoyuan works on the system design, implement specific protocols, and helps with the case study and building the homepage. Tianchen works on the system implementation, protocol packing, and helps with the protocol implementation. Bryce works on the interaction workflow design, system UI, and helps with the system and protocol implementation. Siyuan works on the system testing and deployment, and helps to extend the solution to a cloud setting.

References

[1] Awesome homomorphic encryption. `https://github.com/jonaschn/awesome-he`. 2022-03-10.

[2] Awesome MPC. `https://github.com/rdragos/awesome-mpc`. 2022-03-10.